1. Stereo Matching Costs
    a. I think NC would work better than SSD because SSD uses the differences between pixels on the patches to calculate cost. With different exposure time, the light intensity in the two photos would be quite different, cause pixels that should not look the same to look more similar.

2. Stereo Matching Implementation

    Code for Q2:

```
import cv2
import numpy as np

img_l = cv2.imread("./000020_left.jpg", cv2.IMREAD_GRAYSCALE)
img_r = cv2.imread("./000020_right.jpg", cv2.IMREAD_GRAYSCALE)

patch_size = 5
half_patch = patch_size // 2
remainder = patch_size % 2
search_pattern = 2  # one every 2 pixel

with open('./000020.txt') as file:
    line = file.readline()
    line = line.split()
    x_left = int(float(line[1]))
    x_right = int(float(line[3]))
    y_bot = int(float(line[4]))
    y_top = int(float(line[2]))
    file.close()

with open('./000020_allcalib.txt') as file:
    lines = file.readlines()
    f = float(lines[0].rstrip().split()[1])
    px = float(lines[1].rstrip().split()[1])
    py = float(lines[2].rstrip().split()[1])
    baseline = float(lines[3].rstrip().split()[1])
    file.close()

depth = np.zeros((y_bot - y_top + 1, x_right - x_left + 1))

for x in range(x_left + half_patch, x_right - half_patch):
    print("column{}".format(x))
    for y in range(y_top + half_patch, y_bot - half_patch):  # depth.shape[0]
        location = 0
        ssd = 99999
        for i in range(x_left + half_patch, x_right - half_patch, search_pattern):
            # ssd1 = 0
            # for patch_x in range(-half_patch, half_patch+1):
            #     for patch_y in range(-half_patch, half_patch+1):
            #         ssd1 += (int(img_l[y+patch_y, x+patch_x]) - 
int(img_r[y+patch_y, i+patch_x]))**2

            ssd1 = np.sum(np.square(
                img_l[y - half_patch:y + half_patch + remainder, x - half_patch:x + 
```

```
half_patch + remainder] -
                img_r[y - half_patch:y + half_patch + remainder, i - half_patch:i +
half_patch + remainder]))

            if ssd1 < ssd:
                ssd = ssd1
                location = i

        if (img_l[y, x] - img_r[y, location]) == 0:
            div = 1
        else:
            div = img_l[y, x] - img_r[y, location]
        depth[y - y_top, x - x_left] = (baseline * f) / div
cv2.imwrite('depth.jpg', depth)
img = cv2.cvtColor(img_l, cv2.COLOR_GRAY2BGR)
img = cv2.rectangle(img, (x_left, y_top), (x_right, y_bot), (0, 0, 255), thickness=2)
cv2.imwrite('box.jpg', img)

half_box_x = depth.shape[1] // 2
half_box_y = depth.shape[0] // 2
z = depth[half_box_y, half_box_x]
box_center = [(half_box_x + x_left - px) * z / f, (half_box_y + y_top - py) * z / f,
z]
real_pixels = np.zeros(depth.shape)

for x in range(depth.shape[1]):
    for y in range(depth.shape[0]):
        Z = depth[y, x]
        X = (x + x_left - px) * Z / f
        Y = (y + y_top - py) * Z / f
        if np.linalg.norm(np.array([box_center[0] - X, box_center[1] - Y,
box_center[2] - Z])) <= 380:
            real_pixels[y, x] = img_l[y + y_top, x + x_left]
cv2.imwrite('real_pixels.jpg', real_pixels)

img1 = cv2.cvtColor(img_l, cv2.COLOR_GRAY2BGR)
img1 = cv2.rectangle(img1, (x_left, y_top), (x_right, y_bot), (0, 0, 255),
thickness=2)
a = x_right - int((x_right - px) * z / f)
b = y_top - int((y_top - py) * z / f)
c = x_left - int((x_left - px) * z / f)
d = y_bot - int((y_bot - py) * z / f)
img1 = cv2.line(img1, (x_right, y_top), (a, b), (255, 0, 0), thickness=2)
img1 = cv2.line(img1, (x_left, y_bot), (c, d), (255, 0, 0), thickness=2)
img1 = cv2.line(img1, (x_left, y_top), (c, b), (255, 0, 0), thickness=2)
img1 = cv2.line(img1, (x_right, y_bot), (a, d), (255, 0, 0), thickness=2)

img1 = cv2.line(img1, (a, b), (a, d), (0, 0, 255), thickness=2)
img1 = cv2.line(img1, (c, d), (c, b), (0, 0, 255), thickness=2)
img1 = cv2.line(img1, (a, d), (c, d), (0, 0, 255), thickness=2)
img1 = cv2.line(img1, (c, b), (a, b), (0, 0, 255), thickness=2)
cv2.imwrite('3dlines.jpg', img1)
```

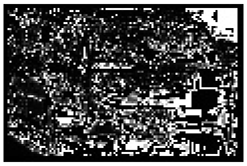
a.

Patch size used is 5x5, sampling method is every other pixel, the matching cost function used is SSD.
SSD is calculated as

```
np.sum(np.square(
                img_l[y - half_patch:y + half_patch + remainder, x -
half_patch:x + half_patch + remainder] -
                img_r[y - half_patch:y + half_patch + remainder, i -
half_patch:i + half_patch + remainder]))
```

at each pixel.
Depth of whole image (I changed my code to bounding box only after because the question only asked pixels within the bounding box):

Depth of pixels within the car bounding box: depth.jpg



There seems to be outliers near the back of the car. Some of the background on the left of the car blends in with the car.

b. I used HD[3] (https://github.com/ucbdrive/hd3) pre-trained model with weights.

Result with HD^3

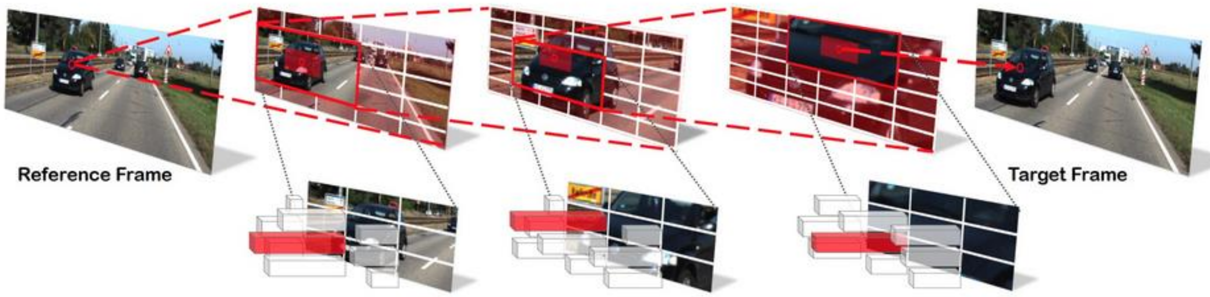
Both the quality and speed are quite different. With my implementation, going through the entire image takes 30 to 40 mins, going through only the car bounding box still takes 2 to 3 mins. With HD^3, the entire image only takes around 10 seconds.

```
[2019-11-24 00:59:48,137 INFO inference.py line 130 15044] => loaded checkpoint './scripts/model_zoo/model.pth'
[2019-11-24 00:59:48,137 INFO inference.py line 141 15044] >>>>>>>>>>>>>>>> Start Test >>>>>>>>>>>>>>>>
[2019-11-24 00:59:57,104 INFO inference.py line 239 15044] <<<<<<<<<<<<<<<<< End Test <<<<<<<<<<<<<<<<<
```

The quality is also quite different. The HD^3 result is gradual and nice looking, it also clearly identifies the car from the background that are further.

c. The model that I picked decomposes the image into different scales hieratically, then it will estimate matching distribution at each scale, in the end the matching distributions get composed back together to form a global match density.

Here are the layers that they use:



For modules, their base layer, level1, level2 are just sequential conv2d. Their level2, level3, level4 each has trees that contain sequential conv2d.

d.

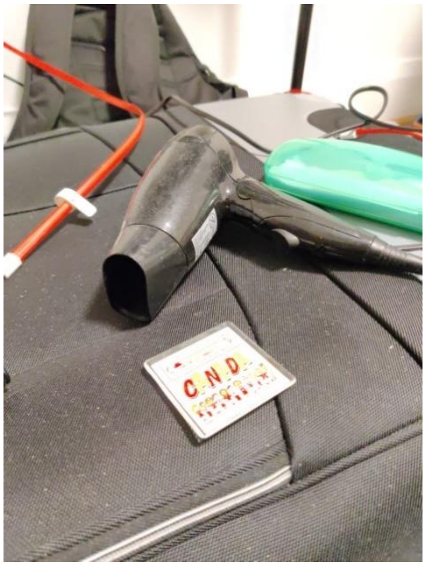
Bounding box: box.jpg

Within the bounding box pixels: real_pixels.jpg

The threshold used for distance is 380.

3D bounding box: 3dlines.jpg

3. Fundamental Matrix



L1

L2

L3

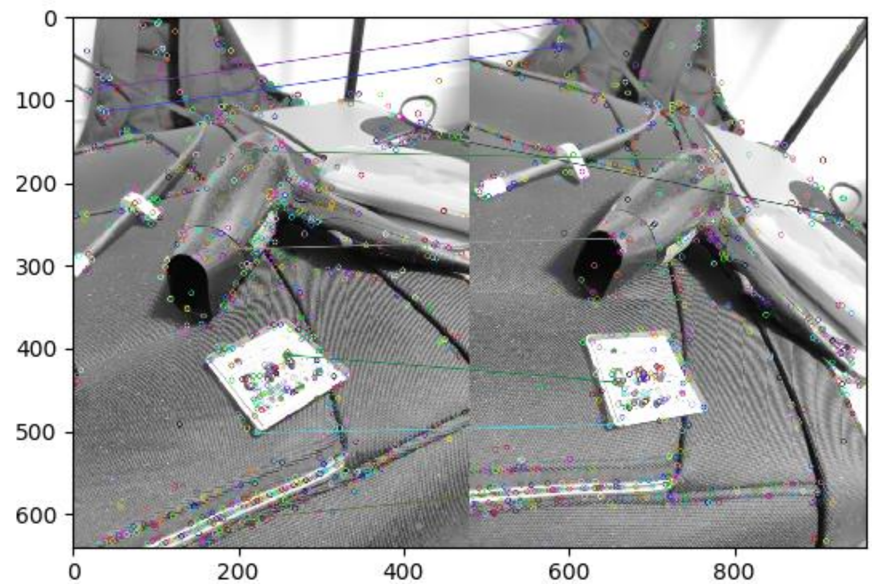
Code for Q3:

```
import numpy as np
import cv2
from matplotlib import pyplot as plt
import cv2.xfeatures2d
import scipy.spatial


def SIFT_matching(img1, img2, threshold):
    if img1[2] is None or len(img1[2]) == 0 or img2[2] is None or len(img2[2]) == 0:
        return cv2.drawMatches(img1[0], img1[1], img2[0], img2[1], [], img2[0], flags=2)
    euclidean = scipy.spatial.distance.cdist(img1[2], img2[2], metric='euclidean')
    sorted1 = np.argsort(euclidean, axis=1)
    closest, closest1 = sorted1[:, 0], sorted1[:, 1]
    left_id = np.arange(img1[2].shape[0])
    dist_ratios = euclidean[left_id, closest] / euclidean[left_id, closest1]
    suppressed = dist_ratios * (dist_ratios < threshold)
    left_id = np.nonzero(suppressed)[0]
    right_id = closest[left_id]
    pairs = np.stack((left_id, right_id)).transpose()
    pair_dists = euclidean[pairs[:, 0], pairs[:, 1]]
    sorted_dist_id = np.argsort(pair_dists)
    sorted_pairs = pairs[sorted_dist_id]
    sorted_dists = pair_dists[sorted_dist_id].reshape((sorted_pairs.shape[0], 1))

    matches = []
    best_8 = np.zeros((8, 2))
    for i in range(len(sorted_pairs)):
        if i < 8:
            best_8[i][0] = sorted_pairs[-i][0]
            best_8[i][1] = sorted_pairs[-i][1]
```

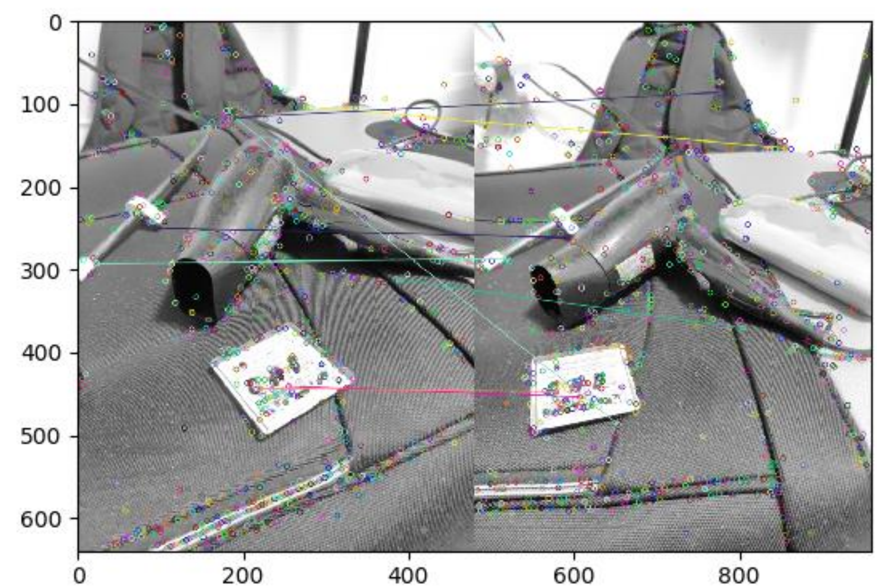
```
        matches.append(cv2.DMatch(sorted_pairs[-i][0], sorted_pairs[-i][1],
sorted_dists[-i]))
    result = cv2.drawMatches(img1[0], img1[1], img2[0], img2[1], matches[:8],
img2[0], flags=2)
    result1 = cv2.drawMatches(img1[0], img1[1], img2[0], img2[1], matches, img2[0],
flags=2)
    return result, result1, best_8


def SIFT_opencv(img):
    sift = cv2.xfeatures2d.SIFT_create()
    kp, des = sift.detectAndCompute(img, None)
    result = cv2.drawKeypoints(img, kp, None)
    return [result, kp, des]


def fundamental_matrix(left, right):
    n = left.shape[0]
    A = np.zeros((n, 9))
    for i in range(n):
        A[i] = [left[i, 0] * right[i, 0], left[i, 1] * right[i, 0], right[i, 0],
               left[i, 0] * right[i, 1], left[i, 1] * right[i, 1], right[i, 1],
               left[i, 0], left[i, 1], 1]

    U, S, V = np.linalg.svd(A)
    F = V[-1].reshape(3, 3)
    U, S, V = np.linalg.svd(F)
    S[2] = 0
    F = np.dot(U, np.dot(np.diag(S), V))
    return F / F[2, 2]


def drawlines(img1,img2,lines,pts1,pts2):
    r,c = img1.shape
    img1 = cv2.cvtColor(img1,cv2.COLOR_GRAY2BGR)
    img2 = cv2.cvtColor(img2,cv2.COLOR_GRAY2BGR)
    for r,pt1,pt2 in zip(lines,pts1,pts2):
        color = tuple(np.random.randint(0,255,3).tolist())
        x0,y0 = map(int, [0, -r[2]/r[1] ])
        x1,y1 = map(int, [c, -(r[2]+r[0]*c)/r[1] ])
        img1 = cv2.line(img1, (x0,y0), (x1,y1), color,1)
        img1 = cv2.circle(img1,(int(pt1[0]),int(pt1[1])),5,color,-1)
        img2 = cv2.circle(img2,(int(pt2[0]),int(pt2[1])),5,color,-1)
    return img1,img2

def findepi(best1, best2, l1, l2, fund_matrix):
    lines2 = cv2.computeCorrespondEpilines(best1.reshape(-1, 1, 2), 1, fund_matrix)
    lines2 = lines2.reshape(-1, 3)
    img3, img4 = drawlines(l2, l1, lines2, best2, best1)

    lines1 = cv2.computeCorrespondEpilines(best2.reshape(-1, 1, 2), 2, fund_matrix)
    lines1 = lines1.reshape(-1, 3)
    img5, img6 = drawlines(l1, l2, lines1, best1, best2)

    plt.subplot(121), plt.imshow(img5)
```

```python
    plt.subplot(122), plt.imshow(img3)
    plt.show()

# a
l1 = cv2.imread("l1.jpg", cv2.IMREAD_GRAYSCALE)
l2 = cv2.imread("l2.jpg", cv2.IMREAD_GRAYSCALE)
l3 = cv2.imread("l3.jpg", cv2.IMREAD_GRAYSCALE)

sift1 = SIFT_opencv(l1)
sift2 = SIFT_opencv(l2)
sift3 = SIFT_opencv(l3)

result_l12, result_l120, best8 = SIFT_matching(sift1, sift2, 0.7)
result_l13, result_l130, best8_1 = SIFT_matching(sift1, sift3, 0.7)
plt.imshow(result_l12)
plt.show()
plt.imshow(result_l13)
plt.show()
# b
best1 = np.zeros((8,2))
best1_1 = np.zeros((8,2))
best2 = np.zeros((8,2))
best3 = np.zeros((8,2))

for i in range(len(best8)):
    best1[i] = sift1[1][int(best8[i][0])].pt
    best2[i] = sift2[1][int(best8[i][1])].pt
    best1_1[i] = sift1[1][int(best8_1[i][0])].pt
    best3[i] = sift3[1][int(best8_1[i][1])].pt

fund_matrix12 = fundamental_matrix(best1, best2)
fund_matrix13 = fundamental_matrix(best1_1, best3)
print("My fundamental matrix")
print(fund_matrix12)
print(fund_matrix13)
# c and d
findepi(best1, best2, l1, l2, fund_matrix12)
findepi(best1_1, best3, l1, l3, fund_matrix13)

# e
F12, mask = cv2.findFundamentalMat(best1, best2, cv2.FM_8POINT)
F13, mask = cv2.findFundamentalMat(best1_1, best3, cv2.FM_8POINT)
print("OpenCv fundamental matrix")
print(F12)
print(F13)
# f
findepi(best1, best2, l1, l2, F12)
findepi(best1_1, best3, l1, l3, F13)
```

a. L1 and L2:



L1 and L3:



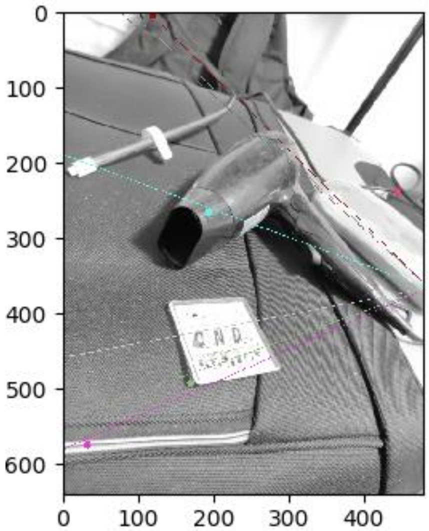
b. L1 and L2 fundamental matrix:

[[5.47703596e-06 1.04911715e-05 -5.10966810e-03]
 [-1.12139250e-05 4.54500960e-06 4.05044154e-03]
 [1.44491285e-03 -6.76338090e-03 1.00000000e+00]]

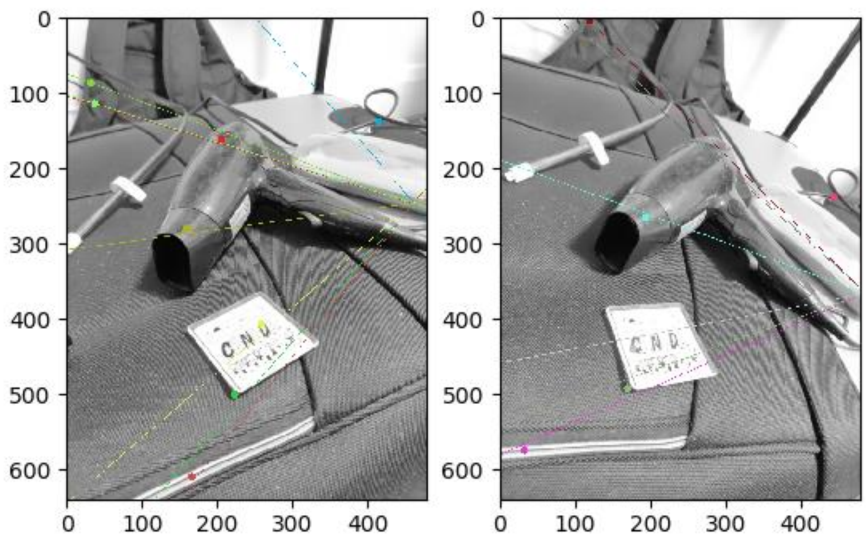
L1 and L3 fundamental matrix:

[[5.77215361e-06 5.73333162e-05 -7.96243972e-03]
 [-5.65007148e-05 3.75514919e-05 5.98267369e-03]
 [7.93461080e-03 -2.05855235e-02 1.00000000e+00]]

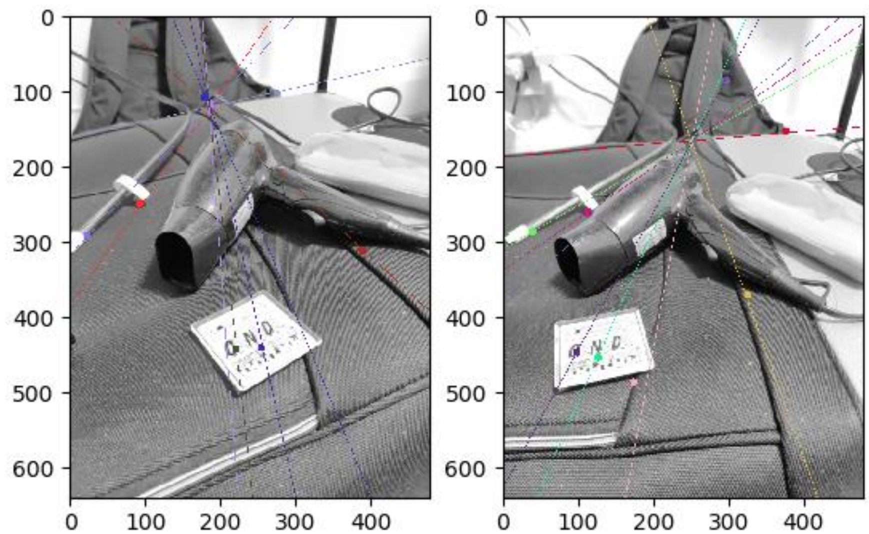
c. L1 and L2 epipolar lines on right image:



d. L1 and L2 epiloar lines on both images:



L1 and L3 epipolar lines on bath images:

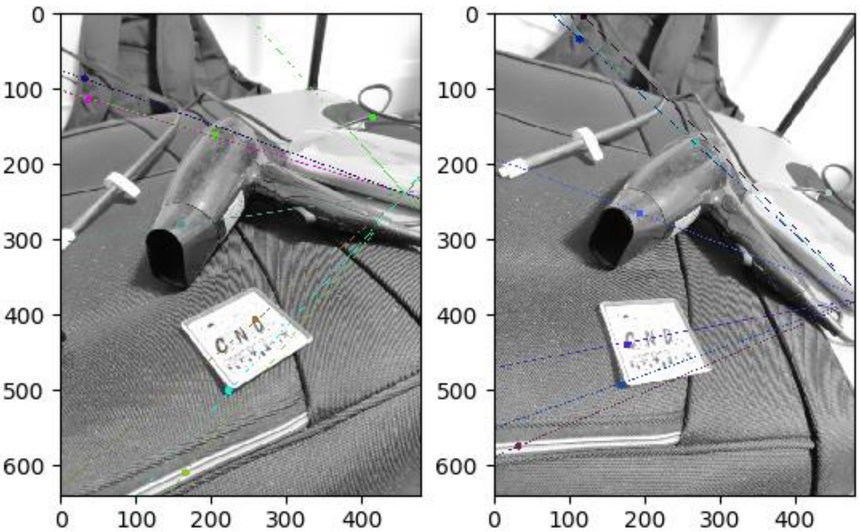


e. OpenCV L1 and L2 fundamental matrix:
[[5.83075519e-06 1.02124310e-05 -5.10687402e-03]
 [-1.10735356e-05 4.40409517e-06 4.00738783e-03]
 [1.31259226e-03 -6.68996062e-03 1.00000000e+00]]

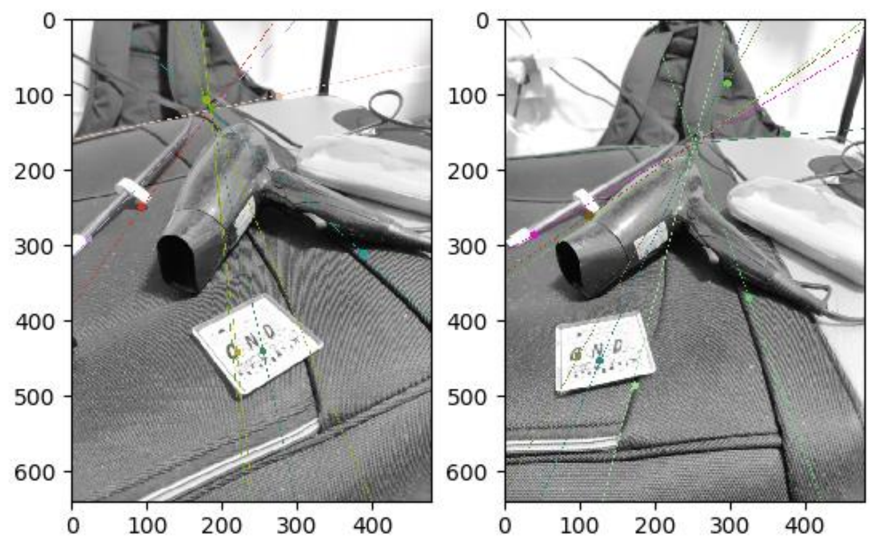
OpenCV L1 and L3 fundamental matrix:
[[5.55528403e-06 5.61202822e-05 -7.78929704e-03]
 [-5.54706302e-05 3.69382125e-05 5.81274795e-03]
 [7.77687451e-03 -2.02426702e-02 1.00000000e+00]]

The first non-zero number after the decimal point is the same for my matrix and OpenCV's. For the next two digits, my matrix and OpenCV's are similar. The other later digits are quite different. The overall number of significant numbers are the same for my and OpenCV's. I think this is because difference in calculation details. OpenCV is implemented in C and converted to Python, while I used a lot of numpy matrix manipulation.

f. L1 and L2 epipolar lines by OpenCV



L1 and L3 epipolar lines by OpenCV



The location of the my calculated epipolar lines are the same as the locations of OpenCV's lines by human eye. I used the same functions to calculate epipolar lines, the the only difference is in the fundamental matrixes, which are very similar.